ESE-RFC 004 Gallagher, Potter, Sgouros, Hankin, Flierl
Category: Proposed Community Standard 2004/08/06
Updates/Obsoletes: None DAP 2.0 Standard

The Data Access Protocol — DAP 2.0

James Gallagher, Nathan Potter, Tom Sgouros, Steve Hankin, Glenn Flierl

Status of this Memo

This is a description of a Proposed ESE Community Standard

Distribution of this Proposed ESE Community Standard is unlimited.

Change Explanation

This RFC does not update or change a previous RFC.

Copyright © NASA, 2004. All Rights Reserved.

Abstract

This document defines the OPeNDAP Data Access Protocol (DAP), a data transmission protocol designed specifically for science data. The protocol relies on the widely used and stable HTTP and MIME standards, and provides data types to accommodate gridded data, relational data, and time series, as well as allowing users to define their own data types.

Contents

1 Introduction

This specification defines the protocol referred to as the Data Access Protocol, version 2.0 (“DAP/2.0”). In this document ‘DAP’ refers to DAP/2.0 unless otherwise noted.

The Data Access Protocol (DAP) is a protocol for access to data organized as name-datatype-value tuples. It is particularly suited to accesses by a client computer to data stored on remote (server) computers which are networked to the client computer. The protocol has been used by the Distributed Oceanographic Data System since 1995[14] and subsequently by many other projects and groups.

While the name-datatype-value model is a nearly universal *conceptual* organization of data, the actual organization of data takes nearly as many forms as there are individual collections because there are many different file formats, APIs and file/directory organizations used to house data. The DAP was designed to hide the implementation of different collections of data behind a simple language-like interface based on the name-datatype-value conceptual model.

1.1 Motivation for Proposing Standardization

The DAP and its associated software components (data servers and client libraries) form the foundation of the National Virtual Ocean Data System (NVODS). NVODS was developed as a system that facilitates access to oceanographic data and data products via the Internet, freeing clients (users) from considerations of: where the data are stored; the format or data management structure under which they are stored; and (to a significant degree) the size of the database. NVODS (formerly known as the ’Virtual Ocean Data Hub’ – VODHub) was created under a 1999 Broad Agency Announcement (BAA) issued by the National Ocean Partnership Program. The concept of the VODHub is to be ”a key element of the full community-based ’system’ to broaden and improve access to ocean data...” The resulting NVODS is also planned for use in the Integrated Ocean Observing System.

Although the DAP was originally developed by and for the oceanographic community it has been adopted by a number of meteorological and climate groups as well and today is extensively used in all three communities - climate, oceanography and meteorology. SEEDS standardization of the DAP will help to accelerate its adoption within these three communities, both through an increase in developers writing to the specification and through an increase in those providing their data via the protocol. This will be of direct benefit to each of the communities individually, and more importantly it will provide the data interoperability required by researchers interested in interdisciplinary problems.

It is important to stress the discipline neutrality of the DAP and the relationship between this and adoption of the DAP in disciplines other than the Earth sciences. First, because the DAP is agnostic as relates to discipline, it can be used across the very broad range of data types encountered in oceanography - biological, chemical, physical and geological. Oceanography may well be unique in this regard, at least within the sub-disciplines of Earth Science. But of particular interest here, is that there is nothing that constrains the use of the DAP to the Earth sciences. For example, groups in the solar physics community have adopted the DAP for their use and proposals are under consideration in other areas of space physics. By standardizing the DAP for the Earth sciences we hope that this will provide an impetus for other disciplines to adopt it as well.

1.2 Requirements

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY” and “OPTIONAL” in this document are to be interpreted as described in RFC 2119 [3].

2 Overall Operation

The DAP is a stateless protocol that governs clients making requests from servers, and servers issuing responses to those requests. This section provides an overview of the requests and responses (*i.e.* the messages) which DAP-compliant software MUST support. These messages are used to request information about a server and data made accessible by that server, as well as requesting data values themselves.

The DAP 2.0 uses HyperText Transfer Protocol (HTTP) as a transport protocol.

The table below provides a description of the DAP messages. The precise details of the requests and responses are described in Section 6 (page 18) and Section 7 (page 21) . A server MUST be able to provide the responses outlined in Table 1. A server MAY support additional request-response pairs.

Table 1: DAP Requests and Responses

Request	**Response**
DDS	DDS or Error
DAS	DAS or Error
DataDDS	DataDDS or Error
Server version	Version information as text
Help	Help text describing all request-response pairs

The DAP uses three responses to represent a data source. Two of these responses, the Dataset Descriptor Structure (DDS) and Dataset Attribute Structure (DAS), characterize the variables, their datatypes, names and atributes. The third response, the Data Dataset Descriptor Structure (DataDDS), holds data values along with name and datatype information.

The DAP returns error information using an Error response. If a request for any of the three basic responses cannot be returned, an Error response is returned in its place.

The three responses (DAS, DDS and DataDDS) are complete in and of themselves so that, for example, the data response can be used by a client without ever requesting either of the two other responses. In many cases, client programs will request the DAS and DDS before requesting the DataDDS, but there is no requirement they do so and no server SHALL require that behavior on the part of clients.

NOTE: The first implementation of the DAP was written in C^{++} and the three basic responses correspond with objects in that implementation. For this reason these responses are referred to as ‘objects’ in some of the DAP documentation. In some cases it is easier to think of these responses as objects and, in those cases, we will use that term in this paper, too. See Section 7 (page 21) for a discussion of the object/response duality.

Operationally, a DAP client sends a request to a server using HTTP. The request consists of a HTTP GET request method, a Uniform Resource Identifier (URI) [2] that encodes information specific to the DAP (see Section 6.1 on page 18) and an HTTP protocol version number followed by a MIME-like message containing various headers that further describe the request. In practice, DAP clients typically use a third-party library implementation of HTTP/1.1 so the GET request, URI and HTTP version information are hidden from the client; it sees only the DAP Uniform Resource Locator (URL) and some of the request headers. The DAP server responds with a status line that includes the HTTP protocol version and an error or success code, followed by a MIME-like message containing information about the response and the response itself. The DAP response is the payload of the MIME-like HTTP response.

In addition to these data objects, a DAP server MAY provide additional “services” which clients may find useful. For example, many DAP-compliant servers provide an HTML-formatted representations of a data source’s structure and a way to get data represented in CSV-style ASCII tables. These additional services are not described in this document, but are instead to be described in ESE Technical Notes.

2.1 Data Representation

Data can be an elusive concept. Data may exist in some storage format on some disk somewhere, on paper somewhere else, in active memory on some server, or transmitted along some wire between two computers. All these can still represent the same data. That is, there is an important distinction to be made between the data and its representation. The data consist of numbers: abstract entities that usually represent measurements of something, somewhere. Data also consist of the relationships between those numbers, as when one number defines a time at which some quantity was measured.

The abstract, platonic existence of data is in contrast to its concrete representation, which is how we manipulate and store it. Data can be stored as BCD numbers in a file on a disk, or as twos-complement integers in the memory of some computer, or as numbers printed on a page. It can be stored in netCDF, HDF, JGOFS, OpenGIS, and any number of other digital storage formats.

The DAP specifies a particular representation of data, to be used in transmitting that data from one computer to another. This representation of some data is sometimes referred to as the “persistent representation” of that data, to distinguish it from the representations used in some computer’s memory. The DAP standard outlined in this document has nothing at all to say about how data is stored or represented on either the sending or the receiving computer. The DAP transmission format is completely independent of these details.

3 Characterization of a Data Source

The DAP characterizes a data source as a collection of variables. Each variable consists of a name, a type, a value, and a collection of *Attributes*. *Attributes*, in turn, are themselves composed of a name, a type, and a value (Section 3.4 on page 11). The distinction between information in a variable and in an *Attribute* is somewhat arbitrary. However, the intention is that *Attributes* hold information that aids in the interpretation of data held in a variable.[1] Variables, on the other hand, hold the primary content of a data source.

3.1 Variables

Each variable in a data source MUST have a name, a type and one or more values. Using just this information and armed with an understanding of the definition of the DAP data types, a program can read any or all of the information from a data source. The names and types of a data source’s variables constitute its *syntactic metadata* [12].

Each variable MAY have one or more *Attributes* associated with it. For information about *Attributes*, see Section 3.4 (page 11) .

The DAP variables come in several different types. There are several *atomic* types, the basic indivisible types representing integers, floating point numbers and the like, and four *constructor* types (also called *container* types) which are flexible collections of other variables. Constructor types may contain both atomic variable types as well as other constructor types.

There is an important distinction to be made here: variables exist in files on a server’s disks in some format, or in a client’s active memory in possibly another format. The DAP does not have anything to say about these formats. The DAP defines, for each data type described in this document, a *persistent representation*, which is

[1] *Attributes* appear in many data storage systems such as netCDF[17], HDF4[15] and HDF5[16]. They also appear under the moniker ‘property’ in Common Lisp[18].

the information actually communicated between a DAP servers and DAP clients. The persistent representation consists of two parts: the declaration of the type and the encoding of its value(s). For a description of the persistent representation see Section 7 (page 21) .

The next two sections describe the abstractions that constitute the variable type menagerie: the range of values and the kind of data each type can represent.

3.2 Atomic variables

As their name suggests, *atomic* data types are indivisible. There are no constraint expression operators that can be used to request part of an instance of one of these types (Section 4 on page 12). Atomic variables are used to store integers, real numbers, strings and URLs. There are three families of atomic types, with each family containing one or more variation:

- Integer
- Floating-point types
- String types

3.2.1 Integer types

The integer types are summarized in Table 2. Each of the types is loosely based on the corresponding data type in ANSI C [10]. However, the DAP, unlike ANSI C, does specify the bit-size of each of the integer types. This is done so that when values are transfered between machines they will be held in the same type of variable, at least within the limits of the software that implements the DAP.

Table 2: The DAP Integer Data types.

name	**description**	**range**
Byte	8-bit unsigned char	0 to $2^8 - 1$
Int16	16-bit signed short integer	-2^{15} to $2^{15} - 1$
Uint16	16-bit unsigned short integer	0 to $2^{16} - 1$
Int32	32-bit signed integer	-2^{31} to $2^{31} - 1$
Uint32	32-bit unsigned integer	0 to $2^{32} - 1$

3.2.2 Floating point types

The floating point data types are summarized in Table 3. The two floating point data types use IEEE 754 [9] to represent values. The two types correspond to ANSI C's `float` and `double` data types.

3.2.3 String types

The two string data types are summarized in Table 4. The first is a simple string type corresponding to the ANSI C notion of a string: a series of US-ASCII characters each represented in a single byte.

Table 3: The DAP Floating Point Data types.

name	**description**	**range**
Float32	IEEE 32-bit floating point [9]	$\pm 1.175494351 \times 10^{-38}$ to $\pm 3.402823466 \times 10^{38}$
Float64	IEEE 64-bit floating point	$\pm 2.2250738585072014 \times 10^{-308}$ to $\pm 1.7976931348623157 \times 10^{308}$

String-type values are limited to 32767 bytes.

The DAP also provides a *URL* data type which is the same as *String* except that it MUST be limited to standard (7-bit) US-ASCII characters, due to the limitations of the syntax of Internet URLs[2], and has the specific meaning of a pointer to some WWW resource.

In DAP applications *URL* is usually used to refer to another data source, in a manner reminiscent of a pointer.

Strings are individually sized. This means that in constructor data types containing multiple instances of some *String*, such as *Sequences* and *Arrays*, successive instances of that *String* MAY be of different sizes.

See Section 7.3.1 (page 30) for other details of the persistent representation of *Strings*.

Table 4: The DAP *String* data types.

name	**description**
String	a series of US-ASCII characters.
URL	a series of US-ASCII characters with the restrictions specified in IETF RFC 2396 [2]

3.2.4 A note regarding implementation of the atomic types

When implementing the DAP, it is important to match information in a data source or read from a DAP response to the *local* data type which best fits those data. In some cases an exact match may not be possible. For example Java lacks unsigned integer types[11]. Implementations faced with such limitations MUST ensure that clients will be able to retrieve the full range of values from the data source. As a practical consideration, this may be implemented by hiding the variable in question or returning an error.

If a variable is automatically hidden (*i.e.* the server analyzes the data source and determines that a particular variable cannot be represented correctly and automatically removes it from those variables made accessible using the DAP) this MUST be noted by adding a global *Attribute* to the data source indicating this has taken place. The note MUST include the name of the variable(s) and the reason(s) for their exclusion. If a variable is removed by a human, this *Attribute* is OPTIONAL.

3.3 Constructor variables

The *constructor* types are assembled from collections of other variables. A constructor type MAY contain both atomic and constructor types. In principle, there are no restrictions on the number of levels or types of nesting of the constructor tyes. However, the *Grid* type imposes some limits on the types it may contain (Section 3.3.3 on page 9).

There are four constructor data types:

- *Array*
- *Structure*
- *Grid*
- *Sequence*

3.3.1 *Array*

An *Array* is a one-dimensional indexed data structure similar to that defined by ANSI C. An *Array*'s member variable MAY be of any DAP data type. *Array* indexes MUST start at zero.

Multidimensional *Arrays* are defined as *Arrays* of *Arrays*. Multi-dimensional *Arrays* MUST be stored in *row-major* order (as is the case with ANSI C). The size of each *Array*'s dimensions MUST be given. The total number of elements in an *Array* MUST NOT exceed $2^{31} - 1$ (2147483647). There is no prescribed limit on the number of dimensions an *Array* may have except that the foregoing limit on the total number of elements MUST NOT be exceeded.

Each dimension of an *Array* MAY also be named.

The number of elements in an *Array* is fixed as that given by the size(s) of its dimension(s).

If you need a data structure which has varying row lengths or an indeterminate number of rows, consider a *Sequence* of *Sequences* or a *Sequence* of *Arrays*. A *Sequence* of *Sequences* can represent data with varying row lengths, and while a *Sequence* of *Arrays* MUST have *Arrays* of the same length in each instance of the *Sequence*, the total length of the *Sequence* is indeterminate. See Section 3.3.4 (page 10) .

3.3.2 *Structure*

A *Structure* groups variables so that the collection can be manipulated as a single item. The *Structure*'s member variables MAY be of any type, including other constructor types. The order of items in the *Structure* is significant only in relation to the persistent representation of that *Structure*.

There is a special case of the *Structure* data type, called *Dataset*. This is the container that encompasses all the variables provided in some data source.

3.3.3 *Grid*

A *Grid* is a special case of a *Structure*, used to supply information to aid in the interpretation of *Arrays*. A *Grid* sets up an association between a target *Array* and a collection of map vectors.

A *Grid* is an association of an N dimensional *Array* with N vectors (*map vectors*), each of which MUST have the same number of elements and the same name as the corresponding dimension of the *Array*. Each vector is used to map indexes of one of the *Array*'s dimensions to a set of values which are normally non-integer (*e.g.* floating point values).

Schematically, a *Grid* is the following:

$$\begin{bmatrix} x_1 & x_2 & x_3 & \cdots & x_m \end{bmatrix}$$

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix} \begin{bmatrix} z_{11} & z_{21} & z_{31} & \cdots & z_{m1} \\ z_{12} & z_{22} & z_{32} & \cdots & z_{m2} \\ z_{13} & z_{23} & z_{33} & \cdots & z_{m3} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ z_{1n} & z_{2n} & z_{3n} & \cdots & z_{mn} \end{bmatrix}$$

Each column of the z *Array* corresponds to an entry in the x map vector, and each row of z corresponds to some y value. So, for example, the data value at $z_{42,33}$ corresponds to the values x_{42} and y_{33}.

For example, a geo-referenced *Grid* might have map vectors that represent the longitude and latitude of each row, so that if you know that the twelfth value of the longitude array is -54, you know that all the values in the twelfth column correspond to longitude 54 degrees west.

The maps MUST be vectors of atomic types.[2]

3.3.4 *Sequence*

A *Sequence* can best be described as an ordered collection of zero or more *Structures*. Each instance in the series consists of the same set of variables, but contains different values.

The semantics of the *Sequence* data type are very close to those of a table in a relational database. You can think of the instances in a *Sequence* as rows in a traditional relational table. OPeNDAP servers that serve data from a DBMS like Oracle or mySQL use *Sequences* to reflect the structure of their data.

A *Sequence* S can be represented as:

$$\begin{array}{cccc} s_{11} & s_{21} & \cdots & s_{n1} \\ s_{12} & s_{22} & \cdots & s_{n2} \\ \vdots & \vdots & & \vdots \\ s_{1i} & s_{2i} & \cdots & s_{ni} \\ \vdots & \vdots & \vdots & \vdots \end{array}$$

Where each $s_1 \cdots s_n$ entry represents a set of DAP variables, and the collection of such entries constitutes the *Sequence*. Every entry of *Sequence* S MUST have the same number, order, and type of variables. If s_{21} is a *Float64*, then all the s_{2i} will also be *Float64* variables. Similarly, in a *Sequence* which contains an *Array* or *Structure*, each instance of the *Array* or *Structure* MUST be the same size. However, a *Sequence* MAY contain a *Sequence* and each instance of the interior *Sequence* MAY have a different number of entries.

Unlike an *Array*, a *Sequence* has no explicit size.

Though the semantics of *Sequences* places limitations on the kinds of requests a client may make of a server, once the *Sequence* has been retrieved, a client program may reference it in any way desired. The DAP defines the persistent representation of data types, and the interaction between client and server (which includes what kinds of requests can be made for what kind of variables), but the DAP does not specify the internal implementation of the data types for any client or server.

[2]This restriction has been put in place to keep writing general clients tractable. If the set of data types in a *Grid*'s map *Arrays* is allowed to be a *Sequence*, for example, any general client would have to be capable of processing that data type in a response. Such a client would be very hard to build.

3.4 Attributes

Attributes are used to associate semantic metadata with the variables in a data source. Attributes are similar to variables in their range of types and values, except that both are somewhat limited when compared to those for variables. Attributes are encoded using the DAS response, and the relationship of that to the DDS response places some extra restrictions on attributes (See Section 7.2.1 on page 23).

Each variable in a data source MAY have *Attributes* associated with it (called *variable attributes*) and the entire *Dataset* (see Section 3.3.2 on page 9) MAY itself have *Attributes*, called *global* Attributes .

While the DAP does not require any particular *Attributes*, some may be required by various *metadata conventions*. The *semantic metadata* for a data source comprises the *Attributes* associated with that data source and its variables [12]. Thus, *Attributes* provide a mechanism by which semantic metadata may be represented without prescribing that a data source use a particular semantic metadata convention or standard.

The data model for *Attributes* is somewhat simpler than that for variables. An *Attribute*'s type MUST either be a *Structure* or one of the atomic types listed below. If the type of the *Attribute* is one of the atomic types, the value MAY be either scalar or one-dimensional *Array*. *Attributes* MAY NOT be multi-dimensional arrays.

If an attribute in a particular data source (*e.g.* an HDF5 file) is a multi-dimension *Array*, it is suggested that the *Attribute* be promoted to a variable and that a new *Attribute* be created for that variable which describes the promotion. This fits the paradigm of remote access better since the multi-dimensional array information would then be accessed with a constraint expression. Since constraint expressions can only be applied to variables, it makes sense to promote such data to a variable.

An *Attribute*'s value MAY be any of the following atomic types:

- `Byte`
- `Int16`
- `UInt16`
- `Int32`
- `UInt32`
- `Float32`
- `Float64`
- `String`
- `URL`

The range of values for atomic type *Attributes* is the same as for the atomic variable types. See Section 7.2.1 (page 23) for information on the persistent representation of atomic-type *Attributes*.

3.5 Attribute Structures

An *Attribute* structure is a container which MAY be empty or which MAY contain atomic type *Attributes* and/or *Attribute* structures. Semantically, an *Attribute* structure is equivalent to the *Structure* variable type; it provides a way to form logical groupings and hierarchies of *Attributes*. An *Attribute* structure MAY NOT directly contain values, only other *Attributes* and *Attribute Structures*.

3.6 Attribute organization

Each variable MUST have an associated *Attribute Structure* and the hierarchy formed by these containers MUST mirror the hierarchy of variables in the data source. There is no requirement that a *Dataset* have an *Attribute Structure* if it has no global *Attributes*. This is one way in which the *Dataset*, which is similar to *Structure*-type variable, is treated specially. All other *Structure* variables are REQUIRED to have an associated *Attribute Structure* (as are ALL variables) but the *Dataset* has no such requirement.

4 Constraint Expressions

A *constraint expression* provides a way for DAP client programs to request certain variables, or parts of certain variables, from a data source. Many data sources are large and many variables from those sources are also large. Often clients are interested in only a small number of values from the entire data source. Constraint expressions provide a way for clients to tell a server which variables, and in many cases, which parts of those variables, they would like.

This section presents the subsampling abilities that MUST be provided by a DAP server. It does so without binding these capabilities to any particular syntax; see Section 6.1.1 (page 19) for the representation of a constraint expression. Some implementations MAY choose to implement additional syntaxes but MUST implement the syntax described there.

Note that an empty constraint expression implies that the entire data source is to be accessed.

4.1 Limiting data by type and by value

A constraint expression provides two different methods to access the information held by a data source. The constraint expression can be used to limit data using the names and/or shapes of variables or by scanning variables and returning only those values that satisfy certain relational expressions. The former are referred to as *projections* while the latter are called *selections*.

A constraint expression MAY combine both projection and selection constraints. For example, a projection might specify that temperatures held in a *Sequence* are to be returned, and a selection would specify that only *Sequence* entries with dates later than 1999 are to be examined. The result returned from a request like this would be a *Sequence* of temperature measurements taken after 1999.

Section 4.1.1 (page 12) describes the projection operations which any DAP implementation MUST support and, likewise, Section 4.1.2 (page 14) describes the required selection operations.

To provide implementors with a means to extend the constraint expression mechanism, it is possible to add functions to a server and to call those as part of the constraint expression. Functions are described in Section 4.1.3 (page 16) .

4.1.1 Projections

The *projection clause* of a constraint expression provides a way to choose parts of a data set based on the shape of the *Dataset* and the variables that it contains. There are two types of projection operations. First, it is possible to choose individual fields of the constructor data types. This is called *field projection* and applies to the *Structure*, *Grid* and *Sequence* data types in the following ways:

Structure A field projection which chooses one or more fields from a *Structure* variable causes a DAP server to return only those named fields from the *Structure*. Note that the *Dataset* itself is similar to a *Structure*. It differs in that it MAY have an attribute container (while all other variables MUST) and it MUST NOT be included in forming fully qualified names (See Section 5 on page 17).

Grid A field projection which chooses one or more fields from a *Grid* variable causes a DAP server to return only those named fields from the *Grid*. It is likely that the variable returned will no longer meet the criteria for a correctly formed *Grid* data type, so the variable may be returned as a *Structure* instead (see Section 4.2 on page 16).

Sequence A field projection which chooses one or more fields from a *Sequence* variable causes a DAP server to return only those named fields from the *Sequence*. For the *Sequence* type, this means returning the N instances but limiting the fields those given in the field projection. For example, suppose the *Sequence* S has P fields:

$$\begin{array}{cccc} s_{11} & s_{21} & \dots & s_{P1} \\ s_{12} & s_{22} & \dots & s_{P2} \\ \vdots & \vdots & & \vdots \\ s_{1N} & s_{2N} & \dots & s_{PN} \end{array}$$

If a field projection is used to choose only the second field, the result of accessing S would be:

$$\begin{array}{c} s_{21} \\ s_{22} \\ \vdots \\ s_{2N} \end{array}$$

When a projection in a constraint expression contains the name of a constructor-type variable, the response MUST include all of the members of that variable. If a projection includes the name of a variable that is not fully qualified (See Section 5 on page 17) the response SHOULD include that variable as if the fully qualified name was given. This provides a shorthand notation for members of a constructor. Suppose there is a *Structure* names `foo` with a member named `bar`. Including `bar` in a constraint expression would cause the `foo.bar` to be included in the response. If a name appears in more than one place in a *Dataset* (for example, suppose a *Grid* is named `SST` and has a member *Array* also named `SST`) the constraint expression evaluator MUST treat the name as fully qualified and include either the matching variable in th response or return an Error response if no variable matches.

When using a field projection, it is possible to request all of the members of a constructor-type variable by using just the name of the constructor.

The second type of projection is a *hyperslab*. A hyperslab is used to limit returned data to those elements that fall within a range of index values, and MAY also specify that the range be subsampled using a *stride*. By including a hyperslab projection for one or more dimensions of a variable it is implied that any unnamed dimensions are to be returned in their entirety.[3] A hyperslab is applied to the *Array*, *Grid* and *Sequence* types in the following way:

Array *Array* dimensions are numbered $0, \dots, N-1$ for an *Array* of rank N. Within each dimension of size M, elements are numbered $0, \dots, M-1$. A hyperslab projection for dimension $n, 0 \leq n < N$ MUST

[3]For some interfaces, it may be necessary to place more restrictions on hyperslab projections.

include the starting index i_{n_s} and ending index i_{n_e} such that $i_{n_s} \leq i_{n_e} \forall \{0 \leq i_n < M\}$. If a stride is included in the hyperslab and is greater than $i_{n_e} - i_{n_s}$ then the hyperslab is equivalent to one where $i_{n_s} = i_{n_e}$ and the original value of i_{n_e} is discarded.

Grid *Grid* dimensions are numbered as are *Array* dimensions; *Grid* dimensions MAY have hyperslab projections applied to them in a manner similar to *Arrays* except that a hyperslab applied to a *Grid* is applied to not only the target array, but also all the corresponding map arrays. For example, given the *Grid*:

$$target = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}, map_1 = \begin{bmatrix} -53 & -52 & -51 & -50 \end{bmatrix}, map_2 = \begin{bmatrix} 26 \\ 25 \\ 24 \\ 23 \end{bmatrix}$$

A hyperslab projection which chose row indexes 1 and 2 and column indexes 1 and 2 would cause a server to return:

$$target = \begin{bmatrix} 6 & 7 \\ 10 & 11 \end{bmatrix}, map_1 = \begin{bmatrix} -52 & -51 \end{bmatrix}, map_2 = \begin{bmatrix} 25 \\ 24 \end{bmatrix}$$

for the *Grid*.

Note that a field and hyperslab projection can be combined for a *Grid* to choose only part of one of the fields, say just part of the the target *Array*. In this case, the hyperslab applied to one field of the *Grid* is equivalent to a hyperslab applied to an *Array*. Effectively, the field projection yields an *Array* and the hyperslab is then applied to that *Array*.

Sequence A hyperslab can be applied to a *Sequence*. A *Sequence* with M instances can have a hyperslab projection applied to it as if it is an *Array* of rank 1. Since the *Sequence* type does not contain an explicit dimension size, the size M is not known until the entire *Sequence* is accessed.[4] A hyperslab projection can be used to ask for the first m elements, the next m elements, etc., which may be very useful for clients which need to know the sizes of varaibles before accessing them. A hyperslab projection for a *Sequence* (i_s, i_e) will return m instances of the *Sequence* such that $m = \lfloor i_e, M - 1 \rfloor - i_s$ depending on whether i_e is an index greater than the number of instances in the *Sequence*. *Sequence* instances are indexed starting with zero.

It is possible to ask for values from several variables in a single constraint expression by including several projections in the constraint expression. Also note that an empty constraint expression, by convention, projects all of every variable in a data source.

4.1.2 Selections

A *selection* provides a way to limit data accessed based on the value(s) of those data. In many ways selections are similar to WHERE claues in SQL[13]. A selection is composed of one or more relational sub-expressions. Each sub-expression MUST be bound to a variable listed in a projection clause. When several sub-expressions constitute a selection, the boolean value of the selection is the logical `AND` of each of the boolean values of each sub-expression. Note that there is no way to perform a logical `OR` operation on the sub-expressions but there is a way, within a sub-expression, to test several values and return `true` if any satisfy the releation.

Each of the relational sub-expressions (*i.e.* relations) is composed of two operands and a relational operator. Each operand MUST be an atomic data type; it MAY be a fully qualified name from the data source or a

Table 5: DAP Selection Relational Operators

Operator	**Meaning**	**Types**
`<`	Less than	Byte, Int16, Int32, UInt16, UInt32, Float32, Float64
`<=`	Less than or equal to	Byte, Int16, Int32, UInt16, UInt32, Float32, Float64
`>`	Greater than	Byte, Int16, Int32, UInt16, UInt32, Float32, Float64
`>=`	Greater than or equal to	Byte, Int16, Int32, UInt16, UInt32, Float32, Float64
`=`	Equal	Byte, Int16, Int32, UInt16, UInt32, Float32, Float64
`!=`	Not equal	Byte, Int16, Int32, UInt16, UInt32, Float32, Float64, String, Url
`=~`	Regular expression match	String, Url

constant. In some cases there are further limitations on the allowed types based on the relational operator. Table 5 lists the operators, their meaning and the data types on which they may be applied.

Operands in a constraint expression selection MAY be either variables in the data source or constants. When constants are used in a selection sub-expression they MAY be either single or multi-valued. If a constant operand has more than one value, each value is used in succession when evaluating the relation. For example, suppose there is a relation:

```
site = {"Diamond_St", "Blacktail_Loop"}
```

Then that relation is true for any instance where `site` is either `Diamond_St` OR `Blacktail_Loop`.

When a variable appears in a selection sub-expression it MUST be single valued.

Selections MAY be applied to the *Sequence* data type in the following way:

Sequence Logically, the relations in a selection bound to a *Sequence* are evaluated once for every instance (*i.e.* row) of the *Sequence*; the result of applying the selection to the *Sequence* is a *Sequence* where all of the instances satisfy all of the relations.

A *Sequence* S with three fields and four instances such as:

$index$	$temperature$	$site$
10	17.2	$Diamond\_St$
11	15.1	$Blacktail\_Loop$
12	15.3	$Platinum\_St$
13	15.1	$Kodiak\_Trail$

A selection such as `index>=` 11 would choose the last three instances:

$index$	$temperature$	$site$
11	15.1	$Blacktail\_Loop$
12	15.3	$Platinum\_St$
13	15.1	$Kodiak\_Trail$

[4]For many *Sequence* variables, it may never be the case that the entire *Sequence* is accessed since it may contain millons of instances.

The selection `site=~ ".*_St"` would choose two instances:

index	*temperature*	*site*
10	17.2	*Diamond_St*
12	15.3	*Platinum_St*

And a selection with the two sub-expressions `index<=11`, site=`~".*_St"` would return only one instance:

index	*temperature*	*site*
10	17.2	*Diamond_St*

4.1.3 Server Functions

A constraint expression MAY also use functions executed by the server. These can appear in a selection or in a projection, although there are restrictions about the data types functions can return.

A function which appears in the projection clause MAY return any of the DAP data types. In this case the return value of the function is treated as if it is a variable present in the top level of the *Dataset* (see Section 3.3.2 on page 9).

A function which appears in the selection clause MAY return any atomic type if it is used in one of the relational sub-expressions. If a function in the selection clause is used as the entire sub-expression, it MUST return an integer value. If that value is zero, the function will evaluate as boolean false, otherwise it will evaluate as boolen true.

When functions encounter an error, a DAP server MUST signal that condition by returning an error response. A server MAY NOT return a partial response; any error encountered while evaluating the constraint expression MUST result in a response that contains an unambiguous error message.

4.2 Data Type Transformation Through Constraints

When a constraint expression has a projection clause that identifies a piece of a constructor variable, such as one field of a *Structure* or just the array part of a *Grid*, the *lexical scoping* of the variable is not abandoned. This is important for avoiding name collisions. For example, if a single item is requestd from a *Structure*, the response MUST contain a *Structure* with only that item.

Here is the behavior for each data type:

Array An *Array* MUST be returned as an *Array* of the same rank as the source *Array* (same number of dimensions). A hyperslab request that effectively eliminates a dimension by reducing its size to 1 does *not* reduce the rank of the returned *Array*. For example, suppose a 10 by 10 element *Array* was subsampled to a 1 by 2 *Array*. The returned variable would still be described as a two dimensional *Array*.

Structure A *Structure* MUST be returned as a *Structure*. If the projection clause of a constraint expression selects only one member of the *Structure*, then a one-member *Structure* MUST be returned. If more than one member of the *Structure* are named in the projection clause, they MUST be returned in the same *Structure*.

Grid A *Grid* modified with a hyperslab operator MUST return another *Grid*, following the same rules as an *Array*. But if the projection clause specifies the elements of the *Grid* independently of one another—the target array, or one of the maps—then a *Structure* is returned containing only the specified variables. A two-dimensional *Grid* named `Cloud` will return a *Grid* in response to a request like this:

`Cloud[1:10][20:30]`. But a request for the target array alone—`Cloud.Cloud[1:10][20:30]`— returns a *Structure* called `Cloud` containing an *Array* called `Cloud`. In this example, the map arrays are not returned.

Sequence A *Sequence* MUST be returned as a *Sequence*, even if a selection clause selects only a single entry or no entry at all. If a projection clause identifies more than one member of the *Sequence*, they MUST be returned in the same *Sequence*.

5 Names

This section describes the persistent representation of names.

A DAP variable's name MUST contain ONLY US-ASCII characters with the following additional limitation: The characters MUST be either upper or lower case letters, numbers or from the set `_ ! ˜ * ' - "` . Any other characters MUST be escaped.

5.1 Escaping characters in names

To escape a character in a name, the character is replaced by the sequence `%<Character Code>` where *Character Code* is the two hex digit code corresponding to the US-ASCII character. Note that the characters `(` and `)` (left and right parenthesis) must be escaped because those are used in the constraint expression syntax and not escaping them makes it impossible to parse certain constraint expressions. Similarly, the `.` (period) character must be escaped when it appears as part of the name of a variable because it is used as the separator between names in a fully qualified name. Thus, not escaping the period would make it impossible to parse certain constraint expressions.

5.2 Constructor variable names

The members of a constructor variable can be individually addressed in the following fashion:

Array Individual *Array* items MUST be addressed with a subscripted expression. For an *Array* named `Temp`, the fourteenth member of the *Array* is referenced as `Temp[13]` (all indexes start at zero). A two-dimensional *Array* is addressed with two subscripts, contained in separate brackets: `SurfaceTemp[13][3]`. See Section 6.1.1 (page 19) .

Structure Members of a *Structure* are addressed by appending the member name to the *Structure* name, separated by a dot (`.`). If the *Structure* `Position` has a member named `Height`, then it is addressed as `Position.Height`. The members of a *Structure* MUST have different names from one another.

Grid The arrays in a *Grid* MAY be referenced in the same fashion as the members of a *Structure*. For a two-dimensional *Grid* named `Cloud`, with one-dimensional map vectors `Latitude` and `Longitude`, a member of a map vector is be addressed like this: `Cloud.Latitude[36]`. This refers to a single latitude value. It is also possible to request part of the target array: `Cloud.Cloud[36][42]`, which will return a single data measurement. The *Grid* itself MAY be addressed like an *Array*: `Cloud[36][42]`, which will return a *Grid* containing the value `Cloud.Cloud[36][42]` along with the two map vectors. See Section 4.2 (page 16) for an explanation of how data types are transformed by constraints.

Sequence A *Sequence* member is addressed in the same fashion as a *Structure*. That is, a time called `Releasedate` of a *Sequence* named `Balloons` is addressed as `Balloons.Releasedate`. But note that unlike a *Structure*, this name references as many different values as there are entries in the `Balloons` *Sequence*. A single entry or range of entries in a *Sequence* MAY be addressed with a hyperslab operator like the items in an *Array*. The variables in a *Sequence* MUST have different names from one another.

5.3 Fully Qualified Names

The lexical scoping rules of the DAP require some description. The important concept is the *fully qualified name* , which is an unambiguous name for some variable or attribute.

5.3.1 Variable Names

The fully qualified name of a variable is composed of the ordered collection of variable names, starting at the *Dataset* level but not including the *Dataset* name, that can be followed to the terminal variable name. The names MUST be separated by the dot (.) character. Thus, if a *Dataset* named `test` contains a structure named `sst` which contains a variable named `foo`, the fully qualified name would be `sst.foo`.

5.3.2 Attribute Names

The fully qualified name of an *Attribute* is composed of the ordered collection of *Attribute* names, starting at the *Dataset* level but not including the *Dataset* name, that can be followed to the terminal source *Attribute*. The names MUST be separated by the dot (.) character. Thus, if a *Dataset* named `test` contains a structure named `sst` which contains a variable named `foo`, the fully qualified name of the *Attributes* of foo would be `sst.foo`. If `foo` possessed an *Attribute* named `fruit` then the fully qualified name for `fruit` would be `sst.foo.fruit`.

NOTE: Forming the fully qualified name for an *Attribute* is largely a formality in DAP/2.0 since it is only possible to request all of the *Attributes*. However, the requirements are included here as a guide. Future versions of the DAP may require its implementation.

6 Requests

The DAP is a client-server protocol: the client makes a request of the server, and the server responds with some information. The request and response travel via HTTP. This section describes the form of requests to servers and responses to clients.

6.1 URL Syntax

A DAP URL is essentially an HTTP URL [5] with additional restrictions placed on the `abs-path` component.

```
DAP-URL        =  "http://" host [ ":" port ] [ abs-path ]
abs-path       =  server-path data-source-id [ "." ext [ "?" query ] ]
server-path    =  [ "/" token ]
data-source-id =  [ "/" token ]
ext            =  "das" | "dds" | "dods"
```

The `server-path` is the pathname to the server, whereas `data-source-id` is the pathname to the data.

The DAP uses HTTP as its session protocol [19], so every DAP URL starts with the scheme `http:`. The `host` and optional `port` name a host and TCP port of an HTTP server that will handle the session. The `host` may also contain authentication information as described in RFC 2617 [6].

The `abs-path` portion of the `DAP-URL` is composed of four parts:

server-path A pathname which identifies the DAP server to handle the request. The servers may be implemented as Common Gateway Interface (CGI) programs or they may use another equivalent scheme (*e.g.* the Apache HTTP daemon's module system).

data-source-id A string passed to the server named by `server-path` that uniquely identifies the source of data on `host`. The `data-source-id` may take the form of a pathname within the HTTP server's document root directory, or it may name the data source in some other way (*e.g.* the DAP server might maintain a table of names mapped to tables in a relational database).

Two special `data-source-id`s MUST be recognized by a DAP server. They are `version` and `help`. When a DAP server receives the `data-source-id` `version` it MUST respond with version information (see Section 7.2.5 on page 28). When a DAP server receives the `data-source-id` `help` it MUST respond with a help message (see Section 7.2.6 on page 29).

ext The optional `ext` part of the `abs-path` tells the DAP server which type of response to return. Each response has a three letter string that is used by the requester. See Section 7 (page 21) for a description of the responses and the `ext` strings used to request them.[5]

query The optional `query` part of the `abs-path` is used with data requests to limit those requests to specific variables or values within the data source. See Section 6.1.1 (page 19) . The `query` part MAY be used with the `dds` and `dods` `ext`.

6.1.1 Constraint expressions

A Constraint Expression (CE) provides a way for clients to request certain variables, or parts of certain variables, from a data source. This section describes the syntax used to encode a constraint expression so that it can be sent, as part of a request, to a server. See Section 4 (page 12) for a general discussion of constraint expressions and the rules for their evaluation.

Some implementations of the DAP MAY choose to provide alternate constraint expression syntax, but all implementations MUST provide the one described here.

Constraint expressions have the following syntax:

```
CE         = [ projection ] *("\&" selection)
projection = variable | variable "," projection
variable   = id | function
function   = id "(" args ")"
args       = arg | arg "," args
arg        = id | quoted-string | integer | float | URL
id         = 1*<any CHAR except CTLs or SP> [ array-dim ]
```

[5]The `ext` is optional because it is possible to request eitehr the version or help response using a special `data-source-id` of `version` or `help`, respectively. See Section 7.2.5 (page 28) and Section 7.2.6 (page 29) .

The constraint expression MUST be encoded using US-ASCII characters. It MAY be used when requesting the DDS or DataDDS (*i.e.* when using the `dds` or `dods` extensions, see Section 7.2.3 on page 27). It MAY NOT be used with the DAS, Version or Help Requests. When it is included in a request, it MUST appear in the request URL as described in Section 6.1 (page 18) . Note that a constraint expression is optional for both the DDS and DataDDS requests; if absent the request if for the entire contents of the data source.

A constraint expression has two parts, the projection and the selection. A projection lists the variables to be returned by the DAP server. If more than one variable is to be returned, then the projection is a comma-separated list of variables. Leaving the projection part of the CE empty is shorthand for requesting all the variables in the data source. A selection is used to request that variables, or instance of variables in the case of a *Sequence*, are returned only if they match certain values. Either or both the projection and selection part of the constraint expression MAY be null.

6.1.1.1 Identifier names

The encoding rules for identifier names are given in Section 5 (page 17) . A valid identifier name MUST appear for `id` in the above grammar. To refer to one field of a constructor type, set `id` to the name of the constructor, followed by a period (`.`) and the field name. To request all of the fields in a constructor, set `id` to the name of the constructor. The `id` value is case-sensitive: the string `temp` is different than the string `Temp`.

6.1.1.2 Hyperslab operators

An *Array* MAY be accessed using only its name to return the entire array or using a hyperslab (`[]`) operator to return a rectangular section of the array. In the later case, the hyperslab is defined for each dimension by a starting index, and ending index, and an optional stride value. An *Array* or *Grid* variable MUST either be unconstrained or have a hyperslab constraint for each of its dimensions. Note that it is possible to combine the syntax that requests a field of a constructor with the *Array* hyperslab syntax to request a section of one of the *Array* variables held in a *Grid*.

```
array-dim = [ start ":" stride ":" stop ]
            [ start ":" stop ]
            [ start ]
start, stride, stop = 1*DIGIT
```

The omitted `stride` value indicates a default of one. If the `stop` is also omitted, its default value is the same as the `start` value. All of these must be integers greater than or equal to zero.

6.1.1.3 Calling server-side functions

Functions MAY be called as part of either the projection or selection clauses. In the case of a selection, the function MUST return a value which can be used when evaluating the clause. In the case of a projection, the function MUST return a DAP variable which will then be the return value of the request or it MUST return nothing in which case it is run for side effect only.

```
selection = *relation | *function
relation  = (id rel-op id) | (value rel-op id)
          | (id rel-op value)
value     = constant | ( "{" 1#constant "}" )
constant  = quoted-string | <int> | <float> | URL
```

6.1.1.4 Syntax errors

Syntax errors in the constraint expression MUST cause an Error response to be returned. The Error response SHOULD contain text that describes the error. The description SHOULD be human readable.

6.2 Request Headers

The headers described in Sections 6.2.1 to 6.2.3 MUST be handled as described. Other headers which are part of HTTP 1.1 MAY be included in the request and MAY be honored by a DAP server.

6.2.1 Accept-Encoding

The `Accept-Encoding` request-header is used by a DAP client to tell a server that it can accept compressed responses. See RFC 2616 [5] for this header's grammar. Values for encodings are `deflate`, `gzip` and `compress`. This header is OPTIONAL. When a client includes this header it is effectively asking the DAP server to encode the response using the given scheme. The server is under no obligation to use the requested encoding. A server MUST NOT use an encoding when a client has not requested it. A client MUST supply the header with every request for which it desires a special encoding.

6.2.2 Host

The `Host` request-header is used by a DAP client to provide its IP address or DNS name to the DAP server. See RFC 2616 [5] for this header's grammar. This header MUST be included with every request.

6.2.3 User-Agent

The `User-Agent` request-header is used by a DAP client to provide specific information about the client software to the DAP server. See RFC 2616 [5] for this header's grammar. This header is RECOMMENDED. DAP servers MAY log this information.

7 Responses

A valid DAP response has the same form as a valid HTTP response. The first line contains the HTTP protocol version, a status code and reason phrase [5]. Following this are the response headers which vary depending on the request and payload of the response (see Section 7.1 (page 21) for a description of the headers). As described in RFC 822 [4], the HTTP response status line and headers are separated from the response's payload by an extra set of CRLF[6] characters which make a blank line.

The ten possible response payloads defined by the DAP are described in detail in Section 7.2 (page 23) .

7.1 Response Headers

The DAP responses use several of the standard MIME headers, in addition to some DAP-specific headers.

[6]The token 'CRLF' is used to denote the carriage return and linefeed characters which correspond to decimal value 10 and decimal vale 13.

7.1.1 Content-Description

The `Content-Description` header is used to tell clients which of the different basic responses is being returned or if an error message is being returned. For any of the basic responses (DDS, DAS, or DataDDS) or the error response, this header MUST be included. This header MUST NOT be included in Version or Help requests. See IETF RFC 2045 [7] for information about this header.

```
Content-Description = "Content-Description :" tag
tag                 = "dods-dds" | "dods-das" | "dods-data" | "dods-error"
```

Example: `Content-Description: dods-error`

7.1.2 Content-Encoding

If a DAP server applies an encoding to an entity, it MUST include the `Content-Encoding` header in the response. See RFC 2616 [5] for this header's grammar.

Example: `Content-Encoding: deflate`

7.1.3 Content-Type

The `Content-Type` header MUST be included in any response from a DAP server. Valid content types for DAP responses are: `text/plain`, `text/html` and `application/octet`.[7] See RFC 2616 [5] for this header's grammar.

Example: `Content-Type: application/octet`

7.1.4 Date

The `Date` header provides a time stamp for the response. This header is needed for servers that support caching. See RFC 2616 [5] for this header's grammar. Servers MUST provide this header.

Example: `Date: Fri, 09 Feb 2001 18:54:55 GMT`

7.1.5 Server

The `Server` header provides information about the server used to process the request. In this case the *server* MAY be either the DAP server or an underlying HTTP server if the DAP server uses that as part of its implementation. See RFC 2616 [5] for this header's grammar. This header is OPTIONAL.

Example:

```
Server: Apache/1.3.12 (Unix)  (Red Hat/Linux) PHP/3.0.15 mod_perl/1.21
```

[7] It would be better to use a multipart document in place of the `application/octet`.

7.1.6 WWW-Authentication

The `WWW-Authenticate` header MUST be included in an HTTP message that has a response code of 401. That is, when the DAP server is asked to provide access to a resource that is restricted and the request does not include authentication information (see “HTTP Authentication: Basic and Digest Access Authentication” [6]). then it must return with a response code of 401 and include the `WWW-Authenticate` header. See RFC 2616 [5] for this header’s grammar.

Example:

```
WWW-Authenticate: Basic realm="special directory, with CGIs"
```

7.1.7 XDODS-Server

The `XDODS-Server` header is used to return DAP server’s implementation version information to the client program.[8] This header MUST be included in every response.

```
XDODS-Server = "XDODS-Server : dods/" version
version      = DIGIT . DIGIT [ . DIGIT ]
```

Example:

```
XDODS-Server: dods/3.2.2
```

7.2 Response Bodies

There are several responses that can come from a server, but four of them are the core functionality of the system. The DAS, the DDS, and the DataDDS can be thought of as data objects containing representations of the data source’s semantic metadata (*i.e.* attributes), its syntactic metadata (structure), and its data, respectively. The Error response MUST ONLY be used to signal problems with a request.

7.2.1 DAS

URL Extension	`das`
Required Headers	`Content-Description: dods-das`
	`Content-Type: text/plain`
	`Server:`
	`Date:`
	`XDODS-Server:`

The DAS response is returned as the payload of a message which MUST have `dods-das` as the value of `Content-Description` and `text/plain` as the value of `Content-Type`. The body of the response contains the persistent representation of the DAS object.

A DAS MUST have a container for each variable in the data source. The hierarchy of containers in a DAS MUST follow the hierarchy of constructor types in the DDS. It MAY contain any number of extra containers.

```
das-doc        = "Attributes" "{" *attribute-cont "}"
attribute-cont = attribute-cont | attribute
attribute      = atomic-decl id 1#value ";"
value          = <float> | <int> | id | quoted-string
```

[8]The version information should be changed to reflect the version of the DAP.

7.2.1.1 Encoding Atomic types Atomic type attributes are encoded as follows: Each attribute has a print representation that consists of the type name followed by the attribute name followed by the value or values. The print representation of the value(s) is determined according to:

1. integers: Each integer value is printed using the base 10 ASCII representation of its value.
2. floating point: Each floating point value is printed using the base 10 ASCII representation of its value. The ouput MUST conform to ANSI C's description of `printf` using the `%g` format specification and the precision is 6.
3. string and URL: Strings and URLs are printed in US-ASCII. If the value of a string contains a space, it must be quoted using double quotes ("). If the value contains a double quote, that MUST be escaped using the backslash (\) character. The backslash character is represented as backslash-backslash (\\).

7.2.1.2 Encoding attribute structures Attribute Structures are encoded be printing the name of the Structure, followed by a curly brace ({), followed by the print representation of all its child attributes followed by a closing curly brace (}).

An example DAS is shown in Figure 1.

7.2.2 DDS

URL Extension	`dds`
Required Headers	`Content-Description: dods-dds`
	`Content-Type: text/plain`
	`Server:`
	`Date:`
	`XDODS-Server:`

The DDS response is returned as the payload of a message which MUST have `dods-dds` as the value of `Content-Description` and `text/plain` as the value of `Content-Type`. The body of the response contains the persistent representation of the DDS object.

The DDS is a textual description of the variables and their names and types that compose the entire data set. The data set descriptor syntax is similar too the variable declaration/definition syntax of C and C^{++}. A variable that is a member of one of the base type classes is declared by by writing the class name followed by the variable name. The type constructor classes are declared using C's brace notation.

```
dds-doc   = "data-source" "{" *type-decl "}" id ";"
type-decl = atomic-decl  | array-decl
            | structure-decl | sequence-decl | grid-decl
```

The `dataset` keyword has the same syntactic function as `structure` but is used for the specific job of enclosing the entire data source even when it does not technically need an enclosing element (because at the outermost level it is a single element such as a structure or sequence).

An example DDS is shown in Figure 2.

Variables in the DAP have two forms. They are either atomic types or constructor types.

```
attributes {
   catalog_number {
   }
   casts {
      experimenter {
      }
      time {
         string units "hour since 0000-01-01 00:00:00";
         string time_origin "1-JAN-0000 00:00:00";
      }
      location {
         lat {
            string long_name "Latitude";
            string units "degrees_north";
         }
         lon {
            string long_name "Longitude";
            string units "degrees_east";
         }
      }
      xbt {
         depth {
            string units "meters";
         }
         t {
            float32 missing_value -9.99999979e+33;
            float32 _fillvalue -9.99999979e+33;
            string history "From coads_climatology";
            string units "Deg C";
         }
      }
   }
}
```

Figure 1: Example Dataset Attribute Response. This example matches the DDS shown in Figure 2. Some of the variables in this fictional data source (*e.g.* `catalog_number`) have no attributes. Note that even though they lack attributes, they still have a matching *Attribute Structure*.

```
dataset {
   int catalog_number;
   sequence {
      string experimenter;
      int32 time;
      structure {
         float64 latitude;
         float64 longitude;
      } location;
      sequence {
         float depth;
         float temperature;
      } xbt;
   } casts;
} data;
```

Figure 2: Example Dataset Descriptor Response.

7.2.2.1 Atomic variables Atomic variables are similar to predefined variables in procedural programming languages like C or Fortran (*e.g.* `int` or `integer*4`).

byte an 8-bit byte;unsigned char in ANSI C.

int16 a 16-bit signed integer.

uint16 a 16-bit unsigned integer.

int32 a 32-bit signed integer.

uint32 a 32-bit unsigned integer.

float32 the IEEE 32-bit floating point datatype (ANSI C's `float`).

float64 the IEEE 64-bit floating point datatype (ANSI C's `double`) .

string a sequence of bytes terminated by a null character.

URL represented as a string, but may be dereferenced in a CE; see Section 4 (page 12) .

```
atomic-decl = atomic-type id ";"
atomic-type = "Byte" | "Int16" | "Uint16" | "Int32" | "Uint32"
                 | "Float32" | "Float64" | "String" | "Url"
id          = (ALPHA | "_" | "%" | "." )
              *(ALPHA | DIGIT | "/" | "_" | "%" | "." )
```

7.2.2.2 Array An **Array** is a one dimensional indexed data structure as defined by ANSI C. Multidimensional arrays are defined as arrays of arrays. The size of each array's dimensions must be given. Each dimension of an array may also be named.

```
array-decl  = array-types id array-dims ";"
array-types = atomic-decl | structure-decl | sequence-decl | grid-decl
array-dims  = array-dim | array-dim array-dims
array-dim   = "[" [ name "=" ] 1*DIGIT "]"
```

The number of dimensions MUST be greater than zero.

7.2.2.3 ***Structure*** A structure groups variables so that the collection can be manipulated as a single item. The variables can be of any type.

```
structure-type  = structure "{" *structure-types "}" ";"
structure-types = atomic-type | array-type
                  | structure-type | sequence-type | grid-type
```

7.2.2.4 **Sequence** A sequence is an ordered set of N variables which has several instantiations (or values). Variables in a sequence may be of different types. Each instance of a sequence is one instantiation of the variables. Thus a sequence can be represented as:

$$\begin{matrix} s_{00} & \cdots & s_{0n} \\ \vdots & & \vdots \\ s_{i0} & \cdots & s_{in} \end{matrix}$$

Every instance of sequence S has the same number, order, and type of variables. Thus in a sequence which contains an array, each instance of the array MUST be the same size.[9] A sequence implies that each of the variables is related to each other in some logical way. A sequence is different from a structure because its constituent variables have several instances while a structure's variables have only one instance (or value).

```
sequence-decl  = sequence "{" *sequence-types "}" ";"
sequence-types = atomic-type | array-type
                 | structure-type | sequence-type | grid-type
```

7.2.2.5 ***Grid*** A grid is an association of an N dimensional array with N named vectors, each of which has the same number of elements as the corresponding dimension of the array. Each vector is used to map indices of one of the array's dimensions to a set of values which are normally non-integral (*e.g.* floating point values). The N (map) vectors may be different types. *Grids* are similar to arrays, but add named dimensions and maps for each of those dimensions.

```
grid-decl = "Grid" "{" "Array:" array-decl "Maps:" 1*array-decl "}" ";"
```

7.2.3 DataDDS

URL Extension	`dods`
Required Headers	`Content-Description: dods-data`
	`Content-Type: application/octet`
	`Server:`
	`Date:`
	`XDODS-Server:`

[9] But a sequence may contain a list and each instance of the list may have a different number of elements. This is because arrays must have their size declared while lists do not.

This response body is the one that returns data to the client. It consists of a copy of the DDS, followed by data in its external representation, described in Section 7.3 (page 29) .

The DataDDS entity is returned as the payload of a message whose `Content-Type` header MUST be `application/octet`.[10] The body of the response contains both text which holds a DDS which describes the variables listed in the request and the values for those variables encoded using XDR[20]. The text DDS and the binary data are separated in the response entity by the literal `Data:`.

```
DataDDS = DDS CRLF "Data:" CRLF *OCTET
```

Clients MAY supply a constraint expression (see Section 4 on page 12) with any `DataDDS` request. The DDS in the `DataDDS` response describes the variables returned. The order that the variables are listed in the DDS MUST match the order of the values in the binary section of the DataDDS response. If the response contains constructor types, then the variables are sent in the order they would be visited in a depth-first traversal of the accompanying DDS.

7.2.4 Error

URL Extension	n/a
Required Headers	`Content-Description: dods-error`
	`Content-Type: text/plain`
	`Server:`
	`Date:`
	`XDODS-Server:`

When a server encounters an error, either in its software or in the users request, it MUST return an error response. The body of the response contains an error code along with text that provides a description of the problem encountered. Server writers are encouraged to provide text that describes the problem with enough information to enable a user to correct the problem or submit a meaningful bug report to the server's maintainer.

```
Error      = "Error" "{" "code=" error-code ";"
                         "message=" error-msg ";" "}"
error-code = 1*DIGIT
error-msg  = quoted-string
```

7.2.5 Version

URL Extension	*none*
Required Headers	`Content-Type: text/plain`
	`Server:`
	`Date:`
	`XDODS-Server:`

The `version` response returns information about the DAP version, server version and may return information about a data source's version. The response may be requested two ways: by using the string `version` as the `data-source-id` or by appending the extension `ver` to the data source name (see Section 6.1 on page 18).

[10]This should be multipart/binary.

```
abs-path       = server-path data-source-id [ "." ext [ "?" query ] ]
server-path    = <name of DAP server>
data-source-id = "version"
```

If a DAP server receives a `version` request, it MUST return DAP version version information. If the request is made using the `ver` extension to a `data-source-id` then the server MUST return the DAP version and server version information. It MAY also return a data source version.

Version information should be returned as plain text in the payload of the response. This version information may be essentially that sane as the information in the XDODS-Server header. The intent is to present users and system maintainers with information about servers that can be used to track down problems or determine if a server can be upgraded to a newer version to fix a particular problem.

```
version-response    = dap-version CRLF server-version
                        [ CRLF data-source-version ]
dap-version         = "Core version:" token "/" version-number
server-version      = "Server version:" token "/" version-number
data-source-version = "Dataset version:" token "/" version-number
token               = 1*<any CHAR except CTLs or separators>
version-number      = 1*DIGIT "." 1*DIGIT "." 1*DIGIT
```

7.2.6 Help

URL Extension	n/a
Required Headers	`Content-Type: text/html`
	`Server:`
	`Date:`
	`XDODS-Server:`

The `help` response MUST be returned when either the server receives a URL with no extension (*i.e.* a URL which asks for no object) or when the `data-source-id` portion of the URL is `help`.

```
abs-path       = server-path data-source-id [ "." ext [ "?" query ] ]
server-path    = <name of DAP server>
data-source-id = "help"
```

The second way of requesting the `help` response is analogous to requesting the `version` response.

The `help` response MUST return an ASCII document which lists the extensions recognized by the server. The response MAY return other information as well.

7.3 Encoding Values

This section describes the external (persistent) representation of values held by a DAP Data Source. This is the way the variables are encoded for inclusion in the DataDDS (see Section 7.2.3 on page 27). This specification should not be understood to dictate the storage of variables in a DAP client or server, in memory or on the disk. What a client does with this data is beyond the scope of this specification, which is only concerned with communicating the values from server to client.

From the point of view of the external representation, it is useful to divide the constructor types into aggregate types and array types, making—with the atomic types—three basic types of DAP variables.

Table 6: The XDR data types used by the DAP as the external representations of base-type variables

Base Type	XDR Type
`byte`	`xdr byte`
`int16`	`xdr short`
`uint16`	`xdr unsigned short`
`int32`	`xdr long`
`uint32`	`xdr unsigned long`
`float32`	`xdr float`
`float64`	`xdr double`
`string`	`xdr string`
`URL`	`xdr string`

7.3.1 Atomic types

The DAP uses Sun Microsystems' XDR protocol [20] for the external representation of all of the atomic type variables. Table 6 shows the XDR types used to represent the various base type variables.

7.3.2 Constructor types

In order to transmit constructor type variables, the DAP defines how the various base type variables, which comprise the constructor type variable, are transmitted. Any constructor type variable may be subject to a constraint expression which changes the amount of data transmitted for the variable (see Section 4 on page 12). For each of the four constructor types these definitions are:

7.3.2.1 Array An array is first sent by sending the number of elements in the array twice.[11] The array lengths are 32-bit integers encoded using `xdr_long`.

Following the length information, each array element is encoded in succession. Arrays of bytes are handled differently than other arrays:

1. An array of bytes: These are encoded as is and are padded to a four-byte boundary. Thus an array of five bytes will be encoded as eight bytes.
2. One-dimensional arrays of all types are encoded by encoding each element of the array in the order they appear. Note that atomic types are encoded as XDR would encode an array. Constructor types are encoded by individually encoding each value as described in this section.
3. Multi-dimensional arrays are encoded by encoding the elements using row-major ordering. Note that atomic types are encoded as XDR would encode an array. Constructor types are encoded by individually encoding each value as described in this section.

[11]This is an artifact of the first implementation of the DAP and XDR. The DAP software needed length information to allocate memory for the array so it sent the array length. However, XDR also sends the array length for its own purposes. The demands of backward compatibility have left it in current implementations. Suggestion: In DAP 2.1, add the XDAP-Version header and use that to signal that this particular problem has been fixed; in DAP 2.1 only send the size once.

```
Array        = length length values
length       = <32-bit integer, signed, big endian>
values       = bytes | other-values
bytes        = <8-bit bytes padded to a four-byte boundary>
other-values = numeric-values | strings | aggregates
```

7.3.2.2 ***Structure*** A structure is sent by encoding each field in the order those fields are declared in the structure. For example, the structure:

```
Structure {
    int32 x;
    float64 y;
} a;
```

Would be sent by encoding the int32 `x` and then the float64 `y`.

Nested structures are sent by encoding their ‘leaf nodes’ as visited in a depth first traversal. For example:

```
Structure {
    int32 x;
    Structure {
        String name;
        Byte image[512][512];
    } picture;
    float64 y;
} a;
```

Would be sent by encoding `x`, then `name`, `image` and finally `y`.

7.3.2.3 ***Sequence*** A Sequence is transmitted by encoding each instance as for a structure and sending one after the other, in the order of their occurrence in the data set. The entire sequence is sent, subject to the constraint expression. In other words, if no constraint expression is supplied then the entire sequence is sent. However, if a constraint expression is given, only the records in the sequence that satisfy the expression are sent

Because a sequence does *not* have a length count, each instance is prefixed by a `start of sequence` marker. Also, to accommodate nested sequences, then end of each sequence as a whole is marked by a `end of sequence` marker.

```
sequence      = instances end-of-seq
instances     = start-of-inst instance-values
end-of-seq    = <byte value 0xA5>
start-of-inst = <byte value 0x5A>
```

7.3.2.4 ***Grid*** A *Grid* is encoded as if it were a *Structure* (one component after the other, in the order of their declaration).

8 Examples

Following are some examples, of requests sent to a server representing some data source, and the response documents returned by those requests.

8.1 Simple request

Assume a server called `server.edu` has some temperature data, stored as a ten-element array named `Tmp`, in a single file called `temp.dat`, in a directory called `data` in the `htdocs` tree. A DAP URL requesting the DDS might look like this:

`http://server.edu/data/temp.dat.dds`

In all of the following examples, carriage returns and new lines are shown as `<CRLF>`. Only shown are the `<CRLF>` characters that are REQUIRED. Since some of all of each response is encoded as text, it makes sense to include extra line breaks to enhance their readability (as we've done here).

The document containing the DDS would look like this:

```
Content-Description: dods-dds<CRLF>
Content-Type: text/plain<CRLF>
Server: Server: Apache/1.3.12 (Unix)  PHP/3.0.15 mod_perl/1.21<CRLF>
Date: Fri, 09 Feb 2001 18:54:55 GMT<CRLF>
XDODS-Server: Friendly-neighborhood DAP implementation v/3.1.1<CRLF>
<CRLF>
Dataset {
  Float32 Tmp[10];
} temp.dat;
```

Note that each of the response headers MUST end in a carriage-return line-feed pair. Also note that a carriage-return line-feed pair on an otherwise blank line MUST separate the response headers from the message body.[7, 8]

The DAS would be requested like this:

`http://server.edu/data/temp.dat.das`

And its response might look like this:

```
Content-Description: dods-das<CRLF>
Content-Type: text/plain<CRLF>
Server: Server: Apache/1.3.12 (Unix)  PHP/3.0.15 mod_perl/1.21<CRLF>
Date: Fri, 09 Feb 2001 18:54:55 GMT<CRLF>
XDODS-Server: Friendly-neighborhood DAP implementation v/3.1.1<CRLF>
<CRLF>
Attributes {
  Tmp {
    Float32 Lat 42.2;
    Float32 Lon -89.3
  }
}
```

The data would be requested like this:

`http://server.edu/data/temp.dat.dods`

The DataDDS containing the data would look like this:

```
Content-Description: dods-data<CRLF>
Content-Type: application/octet-stream<CRLF>
Server: Server: Apache/1.3.12 (Unix)  PHP/3.0.15 mod_perl/1.21<CRLF>
Date: Fri, 09 Feb 2001 18:54:55 GMT<CRLF>
XDODS-Server: Friendly-neighborhood DAP implementation v/3.1.1<CRLF>
<CRLF>
Dataset {
  Float32 Tmp[10];
} temp.dat;<CRLF>
Data:<CRLF>
<Tmp length><Tmp length><value of Tmp[0]> ... <value of Tmp[9]>
```

Where `<Tmp length>` (which appears twice) is the number (32-bit big-endian twos-compliment signed integer) of elements in the array. In this case it would be ten ($\texttt{00 00 00 0A}_{16}$) and `<value of Tmp[0]>`, *et c.*, are the values (32-bit big endian IEEE 754 floating point).

Note that the `Content-Type` header's value is `application/octet-stream` for this type of response and that the the character sequence `<CRLF>Data:<CRLF>` serves as a separator for the response DDS and the binary data values.

The binary data which follows the `<CRLF>Data:<CRLF>` separator MUST NOT contain any carriage-return line-feed pairs.

8.2 *Grid*

Suppose you know that there's a 30 by 50 *Grid* held in some data source at `server.edu`, and you want a 2 by 3 chunk of it. You can request a part of a *Grid* with a constraint expression like this: `grid[20:21][40:42]`.

NOTE: In the remaining examples, we will omit the explicit indication of carriage-return line-feed pairs to simplify presentation.

Ask for the DDS of this data like this:

`http://server.edu/grid-data/grid.dat.dds?grid[20:21][40:42]`

The document containing the DDS would look like this:

```
Content-Description: dods-dds
Content-Type: text/plain
Server: Server: Apache/1.3.12 (Unix)  PHP/3.0.15 mod_perl/1.21
Date: Fri, 09 Feb 2001 18:54:55 GMT
XDODS-Server: Friendly-neighborhood DAP implementation v/3.1.1

Dataset {
  Grid {
    Array:
      Float32 grid[xdimen = 2][ydimen = 3]
    Maps:
      Float32 xdimen[xdimen = 2];
      Float32 ydimen[ydimen = 3];
  } grid;
} temp2.dat;
```

The DAS would be requested like this:

`http://server.edu/grid-data/grid.dat.das?grid[20:21][40:42]`

And its response might look like this:

```
Content-Description: dods-das
Content-Type: text/plain
Server: Server: Apache/1.3.12 (Unix)  (Red Hat/Linux) PHP/3.0.15 mod_perl/1.21
Date: Fri, 09 Feb 2001 18:54:55 GMT
XDODS-Server: Friendly-neighborhood DAP implementation v/3.1.1

Attributes {
  grid{
    String Date "3 Nov 2003, 1433Z";
    String Instrument "Black & Decker Spectrum Analyzer";
  }
}
```

The data would be requested like this:

`http://server.edu/grid-data/grid.dat.dods?grid[20:21][40:42]`

The DataDDS containing the data would look like this:

```
Content-Description: dods-data
Content-Type: text/plain
Server: Server: Apache/1.3.12 (Unix)  PHP/3.0.15 mod_perl/1.21
Date: Fri, 09 Feb 2001 18:54:55 GMT
XDODS-Server: Friendly-neighborhood DAP implementation v/3.1.1

Dataset {
  Grid {
    Array:
      Float32 grid[xdimen = 2][ydimen = 3]
    Maps:
      Float32 xdimen[xdimen = 2];
      Float32 ydimen[ydimen = 3];
  } grid;
} temp2.dat;
Data:
<grid.grid length><grid.grid length>
<grid.grid[0][0]><grid.grid[0][1]><grid.grid[0][2]>
<grid.grid[1][0]><grid.grid[1][1]><grid.grid[1][2]>
<grid.xdimen length><grid.xdimen length><grid.xdimen[0]><grid.xdimen[1]>
<grid.ydimen length><grid.ydimen length><grid.ydimen[0]><grid.ydimen[1]>
<grid.ydimen[2]>
```

The data held in a *Grid* type is encoded as for a *Structure*, one field at a time. In this example, first the `grid.grid` field is encoded, then the `grid.xdimen` and `grid.ydimen`

8.3 *Sequence*

A Sequence of data called `seq` is also stored at `server.edu`. Each record of the sequence contains three values: `xval`, `yval`, and `zval`. A constraint which asks for all values of the *Sequence* where `xval` is less than fifteen would look like:

`xval<15`

Ask for the DDS of these data like this:

`http://server.edu/seq-data/seq.dat.dds?xval<15`

The document containing the DDS would look like this:

```
Content-Description: dods-dds
Content-Type: text/plain
Server: Server: Apache/1.3.12 (Unix)  PHP/3.0.15 mod_perl/1.21
Date: Fri, 09 Feb 2001 18:54:55 GMT
XDODS-Server: Friendly-neighborhood DAP implementation v/3.1.1

Dataset {
  Sequence {
    Int16 xval;
    Int16 yval;
    Int16 zval;
  } seq;
} temp3.dat;
```

The DAS would be requested like this:

`http://server.edu/seq-data/seq.dat.das?xval<15`

And its response might look like this:

```
Content-Description: dods-das
Content-Type: text/plain
Server: Server: Apache/1.3.12 (Unix)  PHP/3.0.15 mod_perl/1.21
Date: Fri, 09 Feb 2001 18:54:55 GMT
XDODS-Server: Friendly-neighborhood DAP implementation v/3.1.1

Attributes {
  xval {
    String units "meters per second";
  }
  yval {
    String units "kilograms per minute";
  }
  zval {
    String units "tons per hour";
  }
}
```

The data would be requested like this:

`http://server.edu/seq-data/seq.dat.dods?xval<15`

The DataDDS containing the data would look like this:

```
Content-Description: dods-data
Content-Type: text/plain
Server: Server: Apache/1.3.12 (Unix)  PHP/3.0.15 mod_perl/1.21
Date: Fri, 09 Feb 2001 18:54:55 GMT
XDODS-Server: Friendly-neighborhood DAP implementation v/3.1.1

Dataset {
  Sequence {
    Int16 xval;
    Int16 yval;
    Int16 zval;
  } seq;
} temp3.dat
Data:
<0x5A><first xval><first yval><first zval>
<0x5A><next xval><next yval><next zval>
<0x5A><next xval><next yval><next zval>
<0x5A><last xval><last yval><last zval><0xA5>
```

A *Sequence*'s values are transmitted one instance at a time. Each instance is prefixed by the *start of instance* marker which is $\texttt{5A}_{16}$. In this example, the constraint `xval<15` causes four instances to be sent and each one is prefixed by the start of instance marker. Once all of the selected instances of the *Sequence* have been sent, the *end of sequence* marker ($\texttt{A5}_{16}$ is written.

Here's a second example of a DataDDS request/response pair for a more complex data source, one that has a *Sequence* within a *Sequence*. The DDS for this data source looks like:

```
Dataset {
  Sequence {
    Float32 lat;
    Float32 lon;
    Sequence {
      Int16 depth;
      Float64 temp;
    } sounding;
  } track;
} temp4.dat;
```

Suppose you wanted to get all the soundings in a lat/lon box that spans the area of 80 to 90 degrees north latitude and 50 to 60 degress west longitude (you would know the units of data source by looking at the attributes which have been omitted from this example). Here's the constraint expression:

```
track.lat>80.0&track.lat<90.0&track.lon>50.0&track.lon<60.0
```

If you requested the DataDDS using the constraint, the response would be:

```
Content-Description: dods-data
Content-Type: text/plain
Server: Server: Apache/1.3.12 (Unix)  PHP/3.0.15 mod_perl/1.21
Date: Fri, 09 Feb 2001 18:54:55 GMT
XDODS-Server: Friendly-neighborhood DAP implementation v/3.1.1

Dataset {
  Sequence {
    Float32 lat;
    Float32 lon;
    Sequence {
      Int16 depth;
      Float64 temp;
    } sounding;
  } track;
} temp4.dat;
Data:
<0x5A><track.lat><track.lon>
<0x5A><track.sounding.depth><track.sounding.temp>
<0x5A><track.sounding.depth><track.sounding.temp>
<0x5A><track.sounding.depth><track.sounding.temp>
<0x5A><track.sounding.depth><track.sounding.temp><0xA5>
<0x5A><track.lat><track.lon>
<0x5A><track.sounding.depth><track.sounding.temp>
<0x5A><track.sounding.depth><track.sounding.temp>
<0x5A><track.sounding.depth><track.sounding.temp><0xA5>
<0x5A><track.lat><track.lon>
<0x5A><track.sounding.depth><track.sounding.temp>
<0x5A><track.sounding.depth><track.sounding.temp>
<0x5A><track.sounding.depth><track.sounding.temp>
<0x5A><track.sounding.depth><track.sounding.temp>
<0x5A><track.sounding.depth><track.sounding.temp><0xA5><0xA5>
```

In this example, the constraint has selected three instances of the outer *Sequence* `track`. For each instance of `track`, there is a complete inner *Sequence* `sounding` which, for this constraint, is sent in its entirety.[12] Note that the end of sequence marker following `<track.sounding.temp>` is the marker for the end of the inner *Sequence*, called `sounding`. The final $\texttt{A5}_{16}$ is the end of sequence marker for the outer *Sequence*, `track`.

References

[Normative References]

[1] ANSI X3.4-1986. Coded character set—7-bit american standard code for information interchange., 1986.

[2] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifiers (URI): Generic syntax. RFC 2396.

[12] You could write a different constraint expression that would choose only values at a certain depth, *et cetera*.

[3] S. Bradner. Key words for use in rfcs to indicate requirement levels. RFC 2119.

[4] David H. Crocker. Standard for the format of arpa internet text messages. RFC 822.

[5] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol — HTTP/1.1. RFC 2616.

[6] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawerence, P. Leach, A. Loutonen, and L.Stewart. Http authentication: Basic and digest access authentication. RFC 2617.

[7] N. Freed and N. Borenstein. Multipurpose internet mail extensions (MIME) part one: Format of internet message bodies. RFC 2045.

[8] N. Freed and N. Borenstein. Multipurpose internet mail extensions (MIME) part two: Media types. RFC 2046.

[9] IEEE 754-1985. IEEE standard for binary floating-point arithmetic, 1985.

[10] ISO/IEC 9899. Programming languages - C, 1999.

[Informative References]

[11] Ken Arnold and James Gosling. *The Java Programming Language*. Addision Wesley, Reading, Massachusetts, 1996.

[12] Peter Cornillon, James Gallagher, and Tom Sgouros. Opendap: Accessing data in a distributed, heterogeneous environment. *CODATA Data Science Journal*, 2:164–174, 2003. Online 5 November, 2003: http://journals.eecs.qub.ac.uk/codata/Journal/contents/2_03/2_03pdfs/DS247.pdf.

[13] C.J. Date. *An Introduction to Database Systems*. Addison Wesley, Reading, Massachusetts, 2000.

[14] James Gallagher and George Milkowski. Data transport within the distributed oceanographic data system. In *World Wide Web Journal: Fourth International World Wide Web Conference Proceedings*, pages 691–702, 1995.

[15] NCSA. HDF 4.1r3 user's guide. http://hdf.ncsa.uiuc.edu/UG41r3_html/, 1999. Retrieved from the World Wide Web 13 October 2003.

[16] NCSA. HDF5 - a new generation of HDF. http://hdf.ncsa.uiuc.edu/HDF5/, 2001. Retrieved from the World Wide Web 15 December 2002.

[17] Russ Rew, Glenn Davis, and Steve Emmerson. *NetCDF User's Guide*. Unidata Program Center, Boulder, Colorado, April 1993. Version 2.3.

[18] Guy L. Steele Jr. *Common Lisp: The Language*. Digital Press, Bedford, Massachusetts, 1984.

[19] W. Richard Stevens. *UNIX Network Programming*. Prentice-Hall, Inc., 2d edition, 1999.

[20] Sun Microsystems, Mountain View, California. *XDR*. Version 4.

Authors

James Gallagher
OPeNDAP, Inc.
165 Dean Knauss Dr.
Narragansett, RI. 02882
Phone: 401.284.1304, email: jgallagher@opendap.org

Nathan Potter
Oregon State University
Phone: 541.737.2293, email: ndp@coas.oregonstate.edu

Tom Sgouros
Manual Writing NA.
Phone: 401.861.2831, email: tomfool@as220.org

Steve Hankin
NOAA PMEL
Phone: 206.526.6080, email: Steven.C.Hankin@noaa.gov

Glenn Flierl
MIT
Phone: 617.253.4692, email: glenn@lake.mit.edu

Appendix A Notational Conventions and Generic Grammar

A.1 Augmented BNF

All of the mechanisms specified in this document are described in both prose and an augmented Backus-Naur Form (BNF) similar to that used by RFC 822 [4]. Implementors will need to be familiar with the notation in order to understand this specification. The augmented BNF includes the following constructs:

`name = definition` The name of a rule is simply the name itself (without any enclosing "<" and ">") and is separated from its definition by the equal "=" character. White space is only significant in that indentation of continuation lines is used to indicate a rule definition that spans more than one line. Certain basic rules are in uppercase, such as SP, LWS, HT, CRLF, DIGIT, ALPHA, etc. Angle brackets are used within definitions whenever their presence will facilitate discerning the use of rule names.

`"literal"` Quotation marks surround literal text. Unless stated otherwise, the text is case-insensitive.

`rule1 | rule2` Elements separated by a bar ("|") are alternatives, e.g., "`yes | no`" will accept `yes` or `no`.

`(rule1 rule2)` Elements enclosed in parentheses are treated as a single element. Thus, "`(elem (foo | bar) elem)`" allows the token sequences "`elem foo elem`" and "`elem bar elem`".

`*rule` The character "`*`" preceding an element indicates repetition. The full form is "`<n>*<m>element`" indicating at least `<n>` and at most `<m>` occurrences of element. Default values are 0 and infinity so that `"*(element)"` allows any number, including zero; `"1*element"` requires at least one; and `"1*2element"` allows one or two.

`[rule]` Square brackets enclose optional elements; `"[foo bar]"` is equivalent to `"*1(foo bar)"`.

`N rule` Specific repetition: `"<n>(element)"` is equivalent to `"<n>*<n>(element)"`; that is, exactly `<n>` occurrences of (element). Thus 2DIGIT is a 2-digit number, and 3ALPHA is a string of three alphabetic characters.

`#rule` A construct `"#"` is defined, similar to `"*"`, for defining lists of elements. The full form is `"<n>#<m>element"` indicating at least `<n>` and at most `<m>` elements, each separated by one or more commas (`","`) and OPTIONAL linear white space (LWS). This makes the usual form of lists very easy; a rule such as `( *LWS element *( *LWS "," *LWS element ))` can be shown as `1#element` Wherever this construct is used, null elements are allowed, but do not contribute to the count of elements present. That is, `"(element), , (element) "` is permitted, but counts as only two elements. Therefore, where at least one element is required, at least one non-null element MUST be present. Default values are 0 and infinity so that `"#element"` allows any number, including zero; `"1#element"` requires at least one; and `"1#2element"` allows one or two.

; **comment** A semi-colon, set off some distance to the right of rule text, starts a comment that continues to the end of line. This is a simple way of including useful notes in parallel with the specifications.

implied `*LWS` The grammar described by this specification is word-based. Except where noted otherwise, linear white space (LWS) can be included between any two adjacent words (token or quoted-string), and between adjacent words and separators, without changing the interpretation of a field. At least one delimiter (LWS and/or separators) MUST exist between any two tokens (for the definition of ”token” below), since they would otherwise be interpreted as a single token.

A.2 Basic Rules

The following rules are used throughout this specification to describe basic parsing constructs. The US-ASCII coded character set is defined by ANSI X3.4-1986 [1].

```
OCTET          = <any 8-bit sequence of data>
CHAR           = <any US-ASCII character (octets 0 - 127)>
UPALPHA        = <any US-ASCII uppercase letter "A".."Z">
LOALPHA        = <any US-ASCII lowercase letter "a".."z">
ALPHA          = UPALPHA | LOALPHA
DIGIT          = <any US-ASCII digit "0".."9">
CTL            = <any US-ASCII control character
                 (octets 0 - 31) and DEL (127)>
CR             = <US-ASCII CR, carriage return (13)>
LF             = <US-ASCII LF, linefeed (10)>
SP             = <US-ASCII SP, space (32)>
HT             = <US-ASCII HT, horizontal-tab (9)>
<">            = <US-ASCII double-quote mark (34)>
```

HTTP/1.1 defines the sequence CR LF as the end-of-line marker for all protocol elements except the entity-body (see appendix 19.3 for tolerant applications). The end-of-line marker within an entity-body is defined by its associated media type, as described in section 3.7.

```
CRLF           = CR LF
```

HTTP/1.1 header field values can be folded onto multiple lines if the continuation line begins with a space or horizontal tab. All linear white space, including folding, has the same semantics as SP. A recipient MAY replace any linear white space with a single SP before interpreting the field value or forwarding the message downstream.

```
LWS            = [CRLF] 1*( SP | HT )
```

The TEXT rule is only used for descriptive field contents and values that are not intended to be interpreted by the message parser. Words of *TEXT MAY contain characters from character sets other than ISO- 8859-1 [22] only when encoded according to the rules of RFC 2047 [14].

```
TEXT           = <any OCTET except CTLs,
                 but including LWS>
```

A CRLF is allowed in the definition of TEXT only as part of a header field continuation. It is expected that the folding LWS will be replaced with a single SP before interpretation of the TEXT value.

Hexadecimal numeric characters are used in several protocol elements.

```
HEX            = "A" | "B" | "C" | "D" | "E" | "F"
               | "a" | "b" | "c" | "d" | "e" | "f" | DIGIT
```

Many HTTP/1.1 header field values consist of words separated by LWS or special characters. These special characters MUST be in a quoted string to be used within a parameter value (as defined in section 3.6).

```
token          = 1*<any CHAR except CTLs or separators>
separators     = "(" | ")" | "<" | ">" | "@"
               | "," | ";" | ":" | "\" | <">
               | "/" | "[" | "]" | "?" | "="
               | "{" | "}" | SP | HT
```

Comments can be included in some HTTP header fields by surrounding the comment text with parentheses. Comments are only allowed in fields containing ”comment” as part of their field value definition. In all other fields, parentheses are considered part of the field value.

```
comment        = "(" *( ctext | quoted-pair | comment ) ")"
ctext          = <any TEXT excluding "(" and ")">
```

A string of text is parsed as a single word if it is quoted using double-quote marks.

```
quoted-string  = ( <"> *(qdtext | quoted-pair ) <"> )
qdtext         = <any TEXT except <">>
```

The backslash character (”\”) MAY be used as a single-character quoting mechanism only within quoted-string and comment constructs.

```
quoted-pair    = "\" CHAR
```

This appendix was copied verbatim from RFC 2616 [5].

Copyright (C) The Internet Society (1999). All Rights Reserved.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this paragraph are included on all such copies and derivative works. However, this document itself may not be modified in any way, such as by removing the copyright notice or references to the Internet Society or other Internet organizations, except as needed for the purpose of developing Internet standards in which case the procedures for copyrights defined in the Internet Standards process must be followed, or as required to translate it into languages other than English.

Appendix B Acronyms and Abbreviations

The following acronyms are used in this text.

BNF Backus-Naur Form

CE Constraint Expression

CGI Common Gateway Interface

DAP Data Access Protocol

DAS Dataset Attribute Structure

DDS Dataset Descriptor Structure

DODS Distributed Oceanographic Data System

DataDDS Data Dataset Descriptor Structure

HTML Hypertext Markup Language

HTTP HyperText Transfer Protocol

MIME Multimedia Internet Mail Extension

SOAP Simple Object Access Protocol

SRS Software Requirements Specification, See IEEE 830–1998

URI Uniform Resource Identifier

URL Uniform Resource Locator

W3C The World Wide Web Consortium, See http://www.w3c.org/

XDR External Data Representation

XML Extensible Markup Language

Appendix C Errata

There are no errata for this document.